

## LA-UR-16-24055

Approved for public release; distribution is unlimited.

Title: ASC/NGC Gitlab Tutorial

Author(s): Junghans, Christoph

Intended for: Report

Issued: 2016-06-10

---

**Disclaimer:**

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.



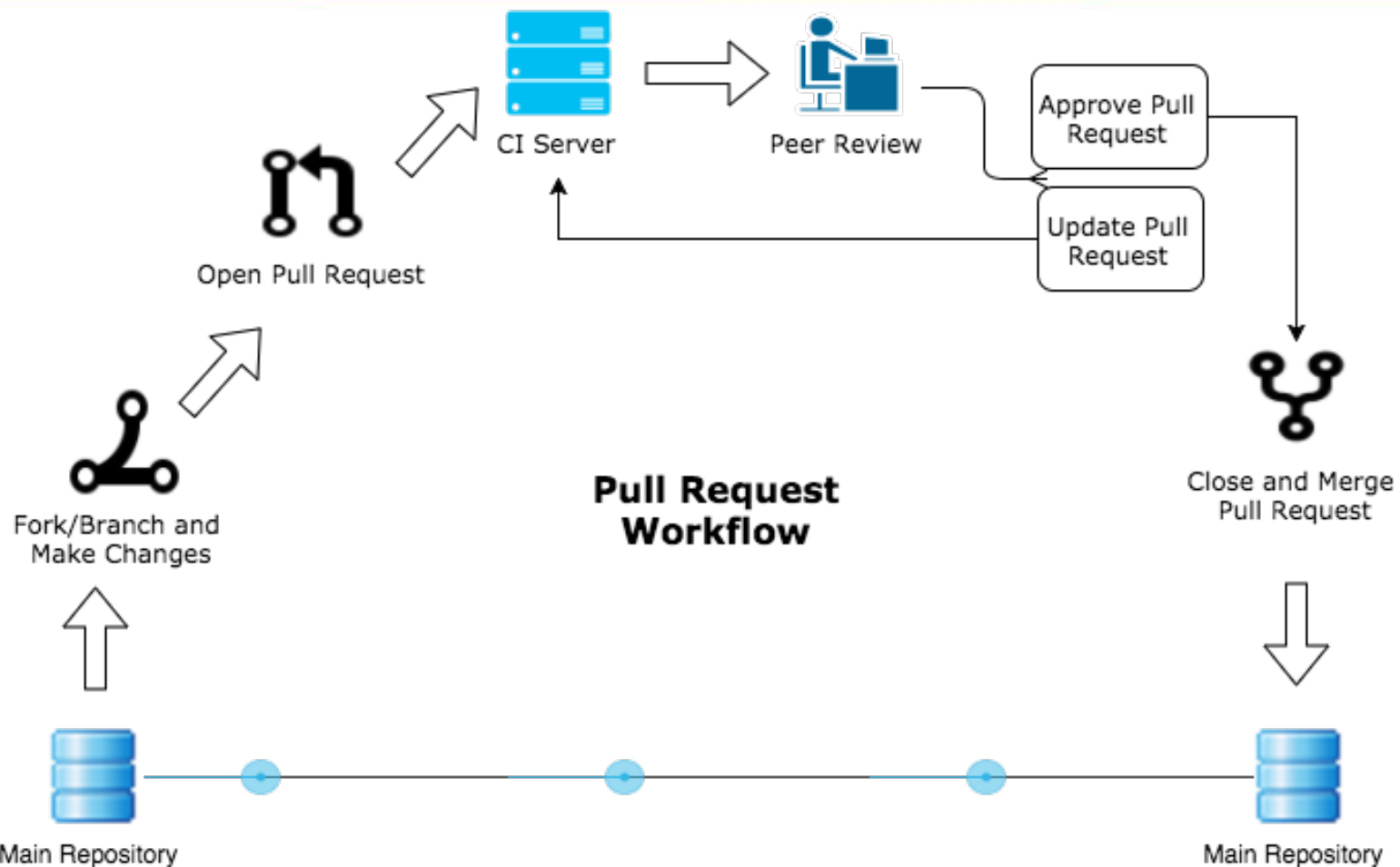
# ASC/NGC Gitlab Tutorial

## \$ man gittutorial{-2} | gitlab.lanl.gov

Christoph Junghans, CCS-7

June 8, 2016

# Pull request Workflow



# Version Control Systems in a Nutshell



VCS manage changes to documents or source code.

Definitions:

- **Repository:** data structure which stores metadata for a set of files and/or directory structure
- **Patch:** unified way to represent a change
- **Commit:** adding a patch (with a message) to the VCS
- **Checkout:** Get a certain version from the repository
- **Diff:** representation of a commit in patch format
- **Revision:** A certain (previous) state of the repository
- **Rollback:** Go back to an older version



UNCLASSIFIED

Slide 3



## Exercise – Manual diffing and patching



- Create a file:  
`"seq 1 10 > file"`
- `"cp file file.old"`
- Edit file
- Create a patch: `"diff -u file file.old > patch"`
- `"rm file.old"`
- Apply patch:  
`"patch -p0 <patch"`
- Look at patchfile
  - Header
  - Line-wise remove and add
- Linus: "We literally used tarballs and patches, which is a much superior source control management system than CVS is."

# Brief VCS history

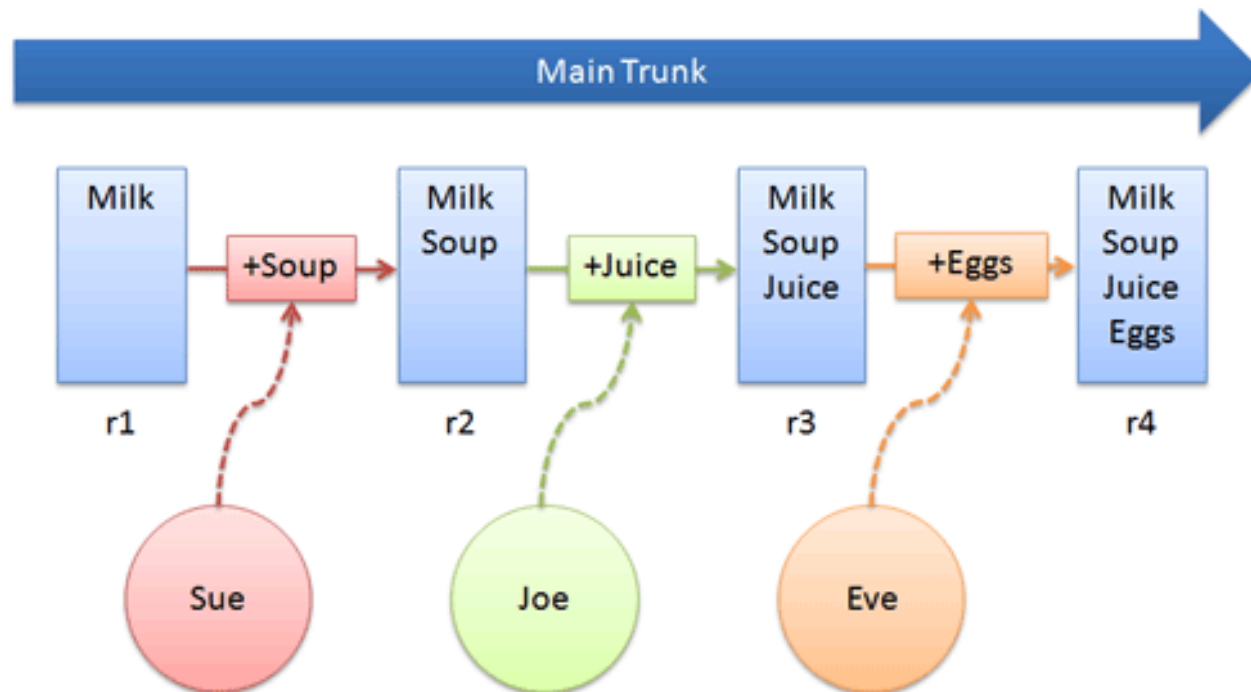


- Version control systems (VCS) have been around for a long time:
  - 1<sup>st</sup> generation: SCCS (72), RCS (82)
  - 2<sup>nd</sup> generation: CVS (90), Subversion (2000)
  - 3<sup>rd</sup> generation:
    - Mercurial/hg (2005)
    - Git (2005)
    - Gnu Arch (2001)
    - Bazaar (2005)
- Mainly two groups: Centralized (CVCS) and distributed (DVCS)

# Centralized Version Control Systems



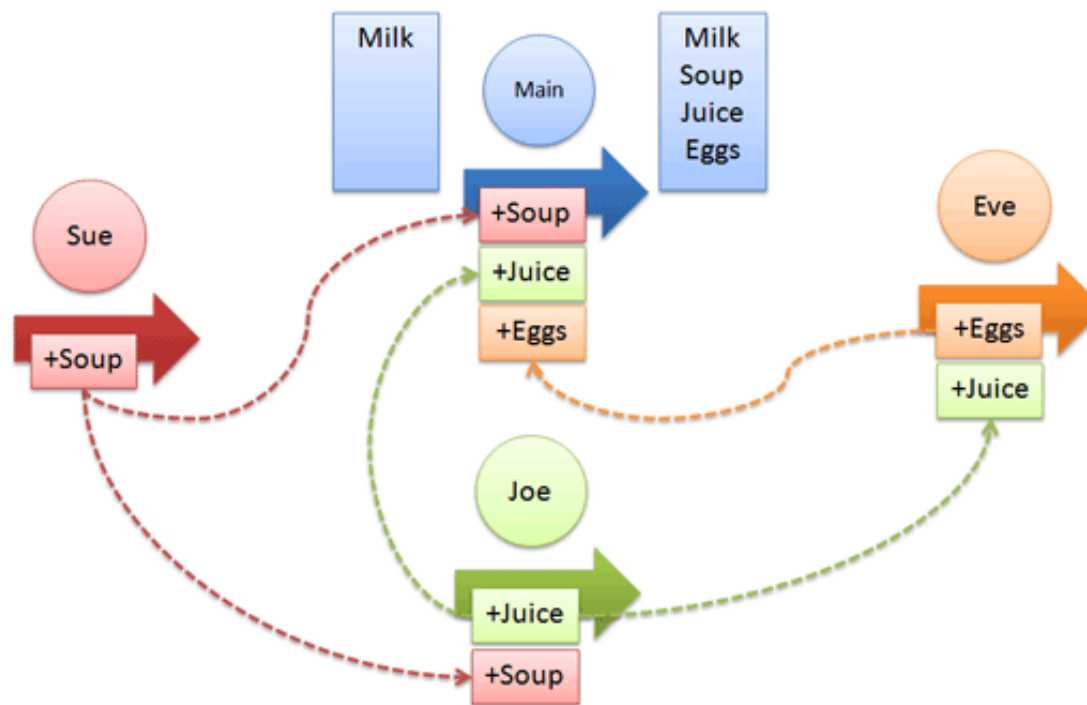
## Centralized VCS



# Distributed Version Control Systems



## Distributed VCS



# CVCS vs. DVCS



## Centralized VCS:

- Checkout contains a single version
- History lives on server
- Commits are versioned
- Backup
- Access via account
- Examples: CVS, SVN

## Distributed VCS:

- Checkout contains all versions
- History lives locally
- Commits are local
- Commits build a graph
- Circle of trust
- Examples: git, hg

# Why git?



- Distributed version control allows better workflows
  - No locking
  - No blocking commits
  - No half-finished feature in the main line
  - Partial merges possible
- Git is the most versatile tool
- Mercurial isn't bad, but poor branching model
- Git vs. Mercurial: Git seems to win the war
- CVS – stop living in the past
- SVN – has its right to exist (big files, restricted access)

# Some words of warning



- Git is different
  - Don't ask, "I did this in CVS, how do I do that in Git."
- Git has about 70 subcommands, I only know 20
  - Don't try to understand every detail
- Git is a pro-tool (like rm), it can destroy data
  - If you did something wrong, don't touch it (like rm)
- Once shared (pushed) commits are hard to contain
  - Think before sharing (like on facebook)
  - Git is never central, even if you try very very hard
- Git's community is not for everyone
  - Friendly alternative: [git@lanl.gov](mailto:git@lanl.gov)

# Making a repository



- "git clone <URL>"
  - URL = path, http{s}, git, scp notation
- "git init" – create empty repo
- Repository = Working directory + ".git" folder
  - There are bare/mirror repos without working directory, mainly for server
- Clone = init + pull + "add remote"
- Exercise:
  - Clone gitlab-course/gitlab-course from gitlab.lanl.gov (use https "-c http.sslVerify=false" option, mind proxy)
  - Init another repo, pull changes from local repo  
"git init repo2", "cd repo2", "git pull ../gitlab-course"

# Git at LANL and on the IC machines



- LANL makes it hard! Custom https certificates and non-transparent proxy setups are so 80's!
  - Stonix update will solve https issue
- Git respects http\_proxy, https\_proxy
  - export http\_proxy=proxyout.lanl.gov:8080
- Git can do ssh hopping
  - echo "ssh wtrw ssh \$@" > ssh\_wtrw
  - export GIT\_SSH=ssh\_wtrw
  - Use ssh whenever you can
  - Fancy version: [https://hpc.lanl.gov/index.php?q=turq\\_scm\\_hints](https://hpc.lanl.gov/index.php?q=turq_scm_hints)
- Use gitlab.lanl.gov for collaborative work inside LANL



# Git setup



- Git supports local and global settings
- Identify yourself
  - "git config (--global) user.name 'J. Robert Oppenheimer' "
  - "git config --global user.email 'rjo@lanl.gov' "
  - Id string will NOT be used for authentication! (Circle of Trust)
  - Exercise:
    - Setup up email, commit name globally
    - Use your secret identity for one of the two repos
    - Have a look at ~/.gitconfig and .git/config
- Use color: "git config --global color.ui true"  
(Pointless in newer versions)

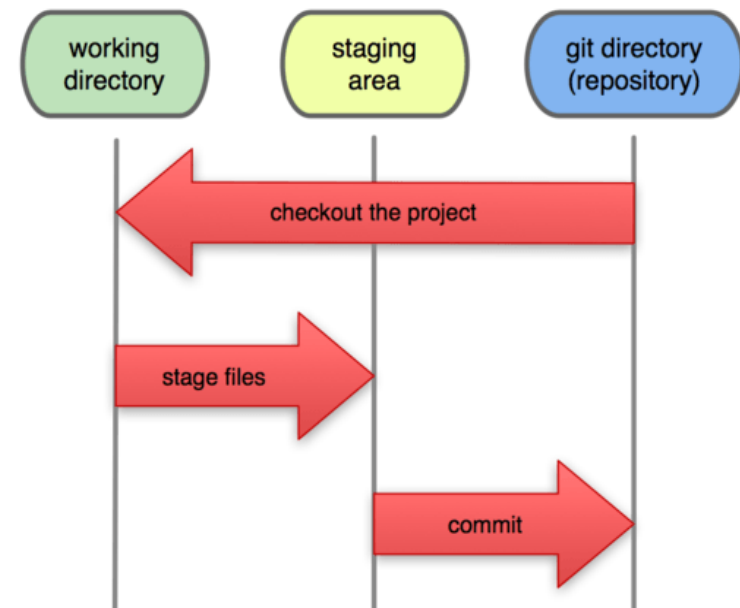
# Making a commit



- Make a change
- Stage files
  - "git add file"
  - "git add -u"
    - Never use "git add -A"
    - "git add -p" to stage parts
- Check status
  - "git status"

"git commit"

## Local Operations



*Commits are local, don't try to make them global by pushing instantaneously:*

*"Commit Often, Perfect Later, Publish Once"*

# Writing a good commit message



- A commit message should be useful!
  - "Test", "Fix a bug", "Make something work" or "Not an empty message" is NOT useful
- Take 30 secs, it will be in logs forever
- Git convention:
  - 1<sup>st</sup> line: summary 50 char
  - 2<sup>nd</sup> line: empty
  - >3<sup>rd</sup> line: details
- Made a mistake? No worries, "git commit --amend" is your friend! (Don't use if commit is already shared.)

## Exercise – Making a commit



- Edit a file (FILE)
- "git diff"
- "git add FILE"
- "git diff --cached"
- "git status"
- "git commit"
- "git log -p"
- Pull that commit into the other repo
- Commit generates a hash that contains all metadata
- Hash is unique
- Commits are local
- Commits are NOT file-based
- Use EDITOR env. variable to change the default editor

# Exchanging commits



- Commit can be shared via push and pull mechanism
  - "pull" means: get a change from someone you trust
  - "push" means: propagating a change to a place you have write permissions to
- Not every single commit needs to be pushed!
- Pulling is preferred (except for bare repos - needs push)
  - Pushing to a normal repo brings workdir out of sync
  - Pulling usually involves a merge (Pull = fetch + merge)
  - Pulling gives control over whom you trust
  - Push and Pull are not symmetric
- Commits can be transferred as files
  - "git format-patch" + "git am"

## Exercise – Mail a patch (more academic)



- Make a commit
- "git format-patch X"  
X=hash of the **parent**  
(look-up in "git log")
- Copy file to the other repo, apply it using "git am <file>"
- "git log --pretty=full"
- Hash has changed, so git won't know it is the same change! - "Think before pushing"
- Committer and author can differ
- Think again, who do you trust
- Hashes can be abbreviated (1<sup>st</sup> 8 characters)

# Branches and logical names (revs)



- Many git commands can act on a hash
  - “git diff HASH”, “git log HASH”
- Using hashes directly is a bit cumbersome
- Logical names (convert to hash: "git rev-parse XXX")
  - HEAD – the latest commit
  - FETCH\_HEAD – fetched hash, ORIG\_HEAD, MERGE\_HEAD
  - Tags, Branch names
  - Ancestors (^/~)
    - HEAD^/HEAD~ - HEAD's 1<sup>st</sup> parent
    - HEAD^^/HEAD~~ - HEAD's 1<sup>st</sup> parent's 1<sup>st</sup> parent
    - HEAD^2 – HEAD's 2<sup>nd</sup> parent (only valid for merges)
    - HEAD^2~ - HEAD's 2<sup>nd</sup> parent's 1<sup>st</sup> parents

# Branches & tags



- Tags are just human-readable names for hashes
  - E.g. "git tag last\_working\_version HEAD^"
  - Tags can be annotated (Author and Message) to be descriptive
  - Tags are local, but can be pushed, pulled
- Branches are just sticky tags
  - Branch name moves with the commit (if HEAD has a name at commit time)
  - Branches are local until shared
- Git supports unnamed Branches, too
- "git branch"
  - Default branch is "master", but not special

## Exercise - Pull from a remote, adding a remote



- Select random host
- Pull changes from that machine
- Check with ssh first
- "git pull user@host:path master" (scp notat.)
- Conflict? Fix it, then "git add -u"
- "git commit"
- Typing user@host:path is very cumbersome
- Add a remote: "git remote add NAME URL"
- "git remote update"
- "git pull NAME master"
- Look at remote branches: "git branch -r" (or "-a")
- Default remote: "origin"



UNCLASSIFIED

Slide 21



# Git vs. Mercurial vs. others



	CVS	SVN	Git	Hg
Distributed			X	X
Centralized	X	X		
Branches	X	X (copies)	X	2X
Tags	X	X	X	X
Web-Server				X
Large Files	X	X	X(ext.)	X (ext.)
Compression			X	X
Global rev#	X (per file)	X		X (per repo)
Keywords	X	X	X (only \$Id\$)	X (ext.)
Commands	30	34	>70	22
Dependencies	C	C++	C, Perl, Bash	C, Python

# More about branches



- Create a branch "git checkout -b NAME START"
  - Use "--track " option to create a branch map
  - Default START hash is HEAD
- "git branch"
  - -d delete: fully merged branch, -D force remove  
(Note: -d/-D just removes the name, not the commits themselves)
  - -m move/ -M force move
- Pushing branches (if not tracked)
  - "git push REMOTE FROM:TO" (names can be crossed)  
(use empty FROM, to remove name on remote)
- Hashes as just unnamed branches



One can use repos in different dirs instead of branches

UNCLASSIFIED

Slide 23



# Diffing branches / revs



- Simple diff of workdir: "git diff REV"
- Diff of missing parts (since last common ancestor)
  - "git diff REV1...REV2" (Mind 3 dots!)
  - "git log REV1...REV2"
- Looking at the graph of merges
  - "gitk"
  - "git log --graph"
  - "git log --decorate --oneline --graph"
- Full diff: "git diff REV1..REV2" (2 dots)
  - "git log --left-right -p REV1..REV2" easier to read

## Exercise – Showing differences



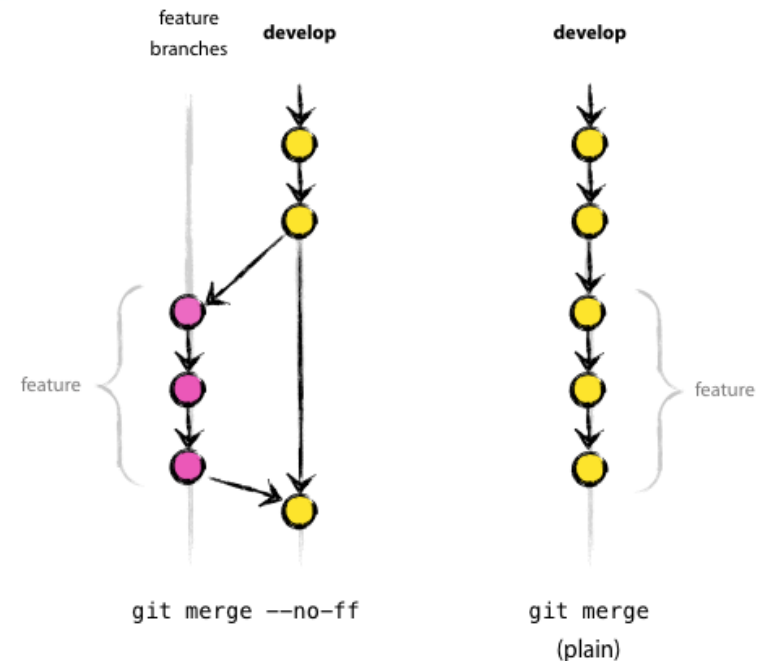
- Select another random host
- Fetch other host's master: "git fetch URL master"
- Find out what differs (use FETCH\_HEAD)
- How many commits have been made?
- Create a new branch "git checkout -b .."
- Commit on the branch
- Switch back to master "git checkout master"
- Try to remove it "git branch -d NAME"
- Merge the branch "git merge NAME"
- Try to remove it again



# Merging revs



- Pull implies a merge
- Git will remember, which parts have been merge before “git branch –contains”
- Git can remember merge resolutions: “git config --global rerere.enabled true” (“reuse recorded resolution”)



*No-fast-forward vs. fast-forward merge.*

# More a about merging



- Git has different merging strategies ("-s" option)
  - "resolve" (default)
  - "recursive" – Suboption (-X) "ignore-all-space" (yeah!)
  - "ours" – "Ok, we will merge your feature" (if we have to)
  - "octopus" – for more than two heads
- Alternatives (changing hashes)
  - Pick a single commit: "git cherry-pick"
  - Use patches "git format-patch"
  - Squashing commits: "git rebase" – very useful on local banch!
- Server repo, one needs to merge before push.



# Branch models



- Branches/Merges are easy: “Best practice is branch out, merge often and keep always in sync”
- Branches can be used for every new feature (development) and removed afterwards
- Linux kernel:
  - Different repos (trees) for different architectures
  - Branch for stable, development, features and next
  - Rebase on local branches before pushing
- Scientific Project
  - Stable, dev, next branch
  - Bug fixes get merged up (stable -> dev)
  - Test etc. live in submodules



UNCLASSIFIED

Slide 28



# Contribution models



- Distributed nature, git allows many contribution models:
  - Email ("git am")
  - Pull
  - Push
- Server - push mechanism
  - Ssh key management using gitolite, keys are in git repo themselves, restricted shell
- Github/Gitlab/Bitbucket as web platform
  - Fork (clone) a repo, make commit, send pull request
  - Pull request involves code review
  - Project policy: Commit gets merged or rebased, squash rebased

# Collaboration using gitlab



- Open-source version of [github.com](https://github.com)
- Instances: [gitlab.lanl.gov](https://gitlab.lanl.gov) and [git.lanl.gov](https://git.lanl.gov)
- Fork and merge model (with review and continuous testing)
- Comes with Issue tracker and wiki
- Cross-project referencing
- Magic commit messages
  - “bla bla (fixes #23)”
  - “bla bla (Related to #23)”

**GitHub**



**GitLab**

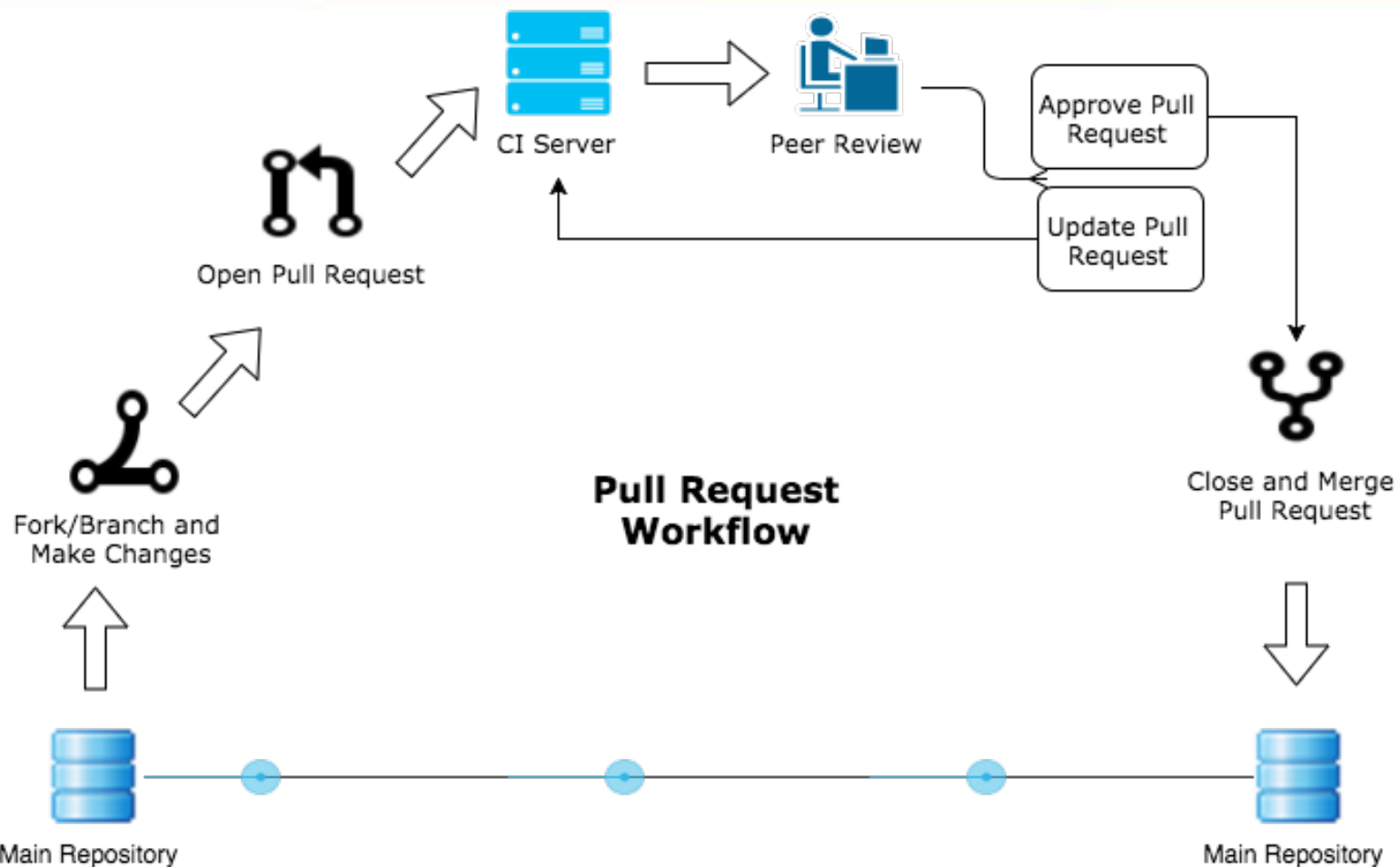
# Exercise: Account Setup



- Create an ssh-key pair: “ssh-keygen”
- Primer on ssh-keys: Public key goes on the server, private key is private
- Go to [gitlab.lanl.gov](https://gitlab.lanl.gov)
- Login using Moniker/Crypto
- “Profile Settings” (on the left)
  - “SSH Keys” (on the left)
  - “Add a new key” (on the top right)
- Paste the content of the public key (.pub file)
- Try connection/setup: `ssh git@gitlab.lanl.gov`



# Pull request Workflow



## Exercise: Fork a repo, push on a branch



- Go to <https://gitlab.lanl.gov/gitlab-course/gitlab-course>
- Fork the repo (by clicking Fork) into your namespace
- Go to your local gitlab-course ( the one with gitlab as origin remote)
- Add your Fork as a remote: “git remote add mygitlab [git@gitlab.lanl.gov:MONIKER/gitlab-course.git](https://gitlab.lanl.gov:MONIKER/gitlab-course.git)”
- Create a local branch: “git checkout -b killer\_feature”
- Try to push the branch to the main repo (will fail): “git push origin killer\_feature”
- Push to your fork: “git push mygitlab killer\_feature”

# Create a pull request



- Go to your fork on gitlab.lanl.gov
- “Merge Requests” (on the left)
- “New Merge Request” (top right)
- Pick your newly create branch (e.g. “killer\_feature”) as source branch
- Look a the diff
- Write a merge request message
- Tag Ondrej on it (@certik)
- Open an issue complain about Ondrej’s slowness in reviewing and assign it to Christoph

# Code Review



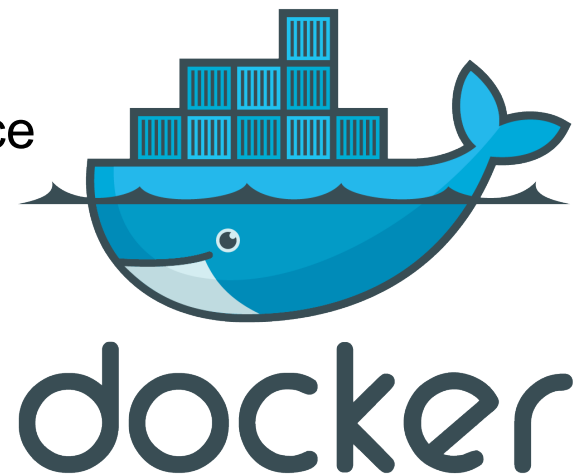
- A code review has two parts
  - Auto-mated testing
  - Human interaction
- Don't break tests!
- Be open for critics (these poor guys have to maintain your crap till the end of time)
- Write Documentation now
- Don't be shy, the sole purpose of Code Review is to improve code quality.
- Don't create to large pull request, one feature at the time



# Behind the Scenes



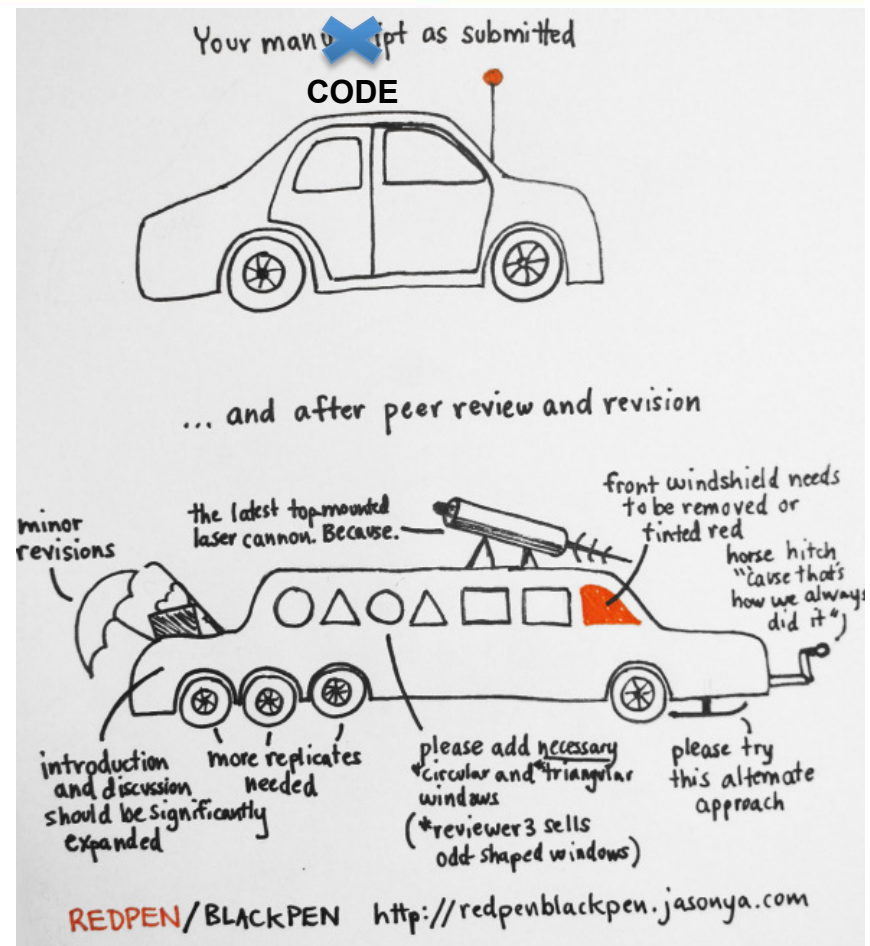
- CI in gitlab is based on docker
- Docker is like fakeroot, but in cool & safe
- Kernel feature
  - Newer kernels can run docker in userspace
  - For older kernel use double virtualization workaround
- Basically no overhead for virtualization
- User has “root” inside the container to e.g. install packages



# Exercise: Make your reviewer happy



- Add another commit to your branch as per Ondrej's suggestion and refer to an issue in commit message
- Push it to your fork
- Check if pull-request got updated
- Ask Ondrej to review again!



# Checking out pull requests



- Pull requests can be checked out and tested locally:
  - “git fetch origin merge-requests/NR/head &&  
git checkout FETCH\_HEAD”  
(merge-requests/NR/head is a special reference name in gitlab)
- Can be make an alias (put this in your ~/.gitconfig)

[alias]

pr-github = !sh -c 'git fetch origin pull/\$1/head:pr-\$1 &&  
git checkout pr-\$1' -

pr-stash = !sh -c 'git fetch origin pull-requests/\$1/from:pr-\$1 &&  
git checkout pr-\$1' -

pr-gitlab = !sh -c 'git fetch origin merge-requests/\$1/head:pr-\$1  
&& git checkout pr-\$1' -

Use “git pr-github”, “git pr-stash”, “git pr-gitlab”

## Other git pearls



- Auto-add files: "git commit -a"
- Stash changes away without committing: "git stash"
- Grep from certain version: "git grep"
- Show files, part of git in workdir: "git ls-files"
- Who introduced that bug? "git blame FILE"
- Launch a web-server: "git instaweb"
- Clean up repo: "git clean", together with .gitignore
- Check repo "git fsck"
- Shallow clones: "git clone --depth=X"

# Other cool git things



- Git hooks
- Git attributes
- Git submodules, bisect
- Git sparse checkout
- git annex (Hello, haskell fans!)
- Gerrit Code-Review
- git fs & git lfs
- Github
- Git's interface to mediawiki



Git as CVS/SVN client

UNCLASSIFIED

Slide 40



## More resources



- "man gittutorial{,-2}"
- git XXX --help | grep Examples
- Video: <https://youtu.be/ZDR433b0HJY>
- **Game:** <http://pcottle.github.com/learnGitBranching/>
- Simple: <http://rogerdudler.github.io/git-guide/>
- For Ruby friends: <http://gitimmersion.com/>
- Everything: <http://git-scm.com/doc>
- Professional: <https://www.udacity.com/course/ud775>
- Graphical: <http://onlywei.github.io/explain-git-with-d3/>



# Rebasing and other backup slides

# Changing history

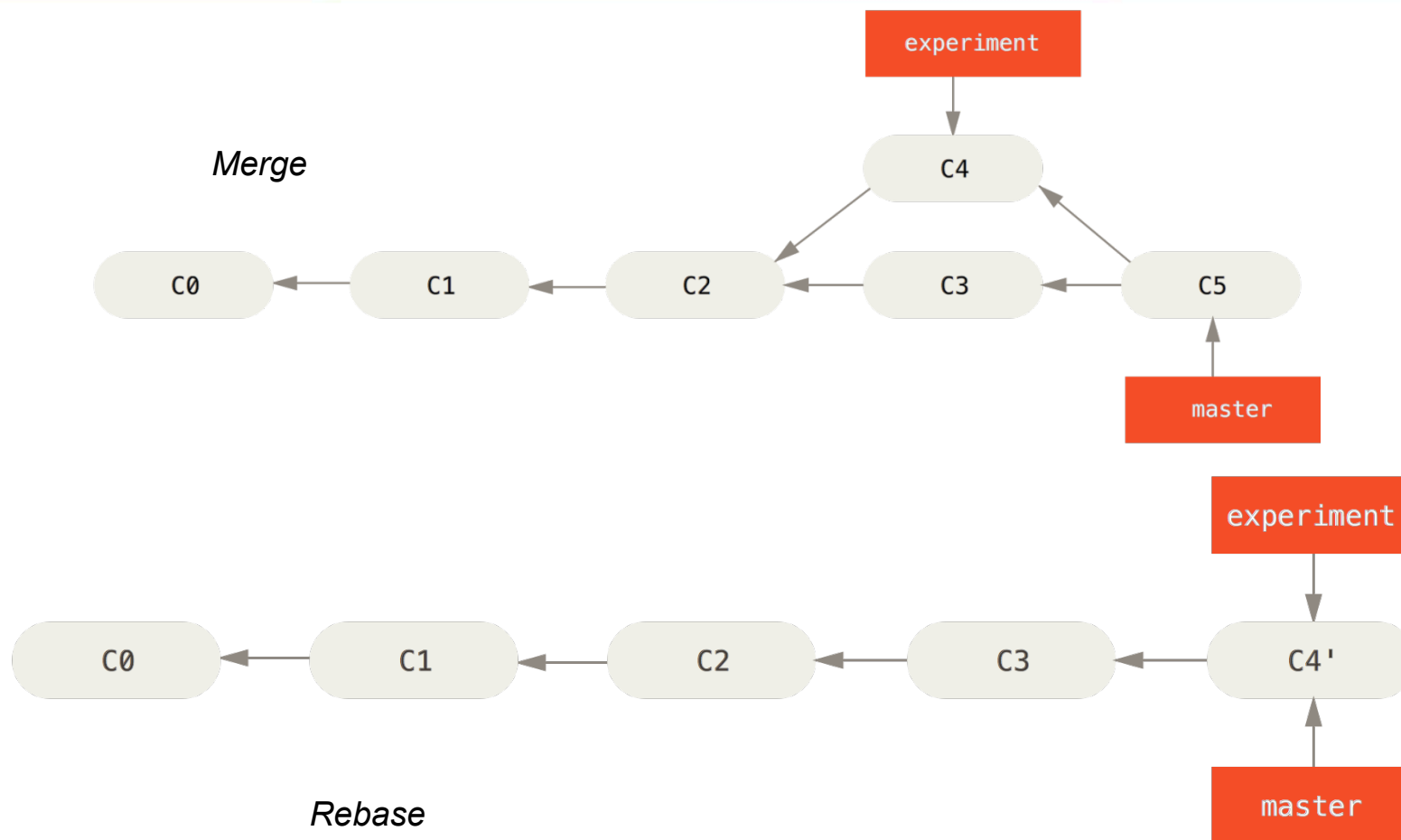


- Rebase means something like: re-apply the commit instead of merging it. (different hash)
- Git provides the very powerful rebase command
- Safest way to use "git pull --rebase"



*"We have to go back", but changing the past is very dangerous!*

# Rebase vs. Merge



# Rebase in practice



- Rebase during pull  
"git pull --rebase"
- Explicit rebase  
"git rebase BASE"  
(won't merge)
- Implicit rebase and squash:  
"git commit --amend"
- Reworking a branch:  
"git filter-branch"



*Pushing rewritten branches: "Just one command away from deleting years of work – you have been warned!"*

## Exercise – Rebase a branch



- Make a commit on master
- Create a branch starting from HEAD^
- Make some commits
- Rebase new branch on master:  
"git pull --rebase . master"
- Run gitk --all
- See if the commit msg shows up twice
- Compare SHA
- Old SHA is still there  
"git log ORIG\_HEAD"
- Rebasing can break correctness of intermediate commits!

# Undoing things



- Undo edit: "git checkout -- file"
- Undo add: "git reset HEAD -- file"
- "git commit --amend"
- Revert a commit by applying the reverse: "git revert"
- "git reset" <REV>
  - "--hard" go back, useful to move branch heads touches both workdir and stage
  - "--soft" bring back pre-commit stage touches nothing (workdir and stage), just undo commit
  - "--mixed" (default), resets the stage but not the working tree
- Squash: "git commit --fixup" + "git rebase --autosquash"

## Exercise – Squash a branch into one commit



- Tag your current HEAD
- Do some commits with: "git commit --squash=TAG" (or --fixup=TAG)
- Rebase into a single commit onto TAG using: "git rebase -i --autosquash TAG^"
- Try: "gitk some\_old\_hash"
- Remove TAG "git tag -d TAG"
- Run "git gc --prune=all"
- Old hashes are gone
- Lost track, check git's blackbox: "git reflog"